TUTORIAL-3

INSERTION SORT:

It sequentially processes a list of records. Each record is inserted in turn at the correct position.

Best case complexity is O(n) and it occurs when the input array is already sorted. Because in this case always the left side element is smaller than the right side element (since the array is sorted) the inner while loop of the insertion sort algorithm (page 47 of course book) will not execute any time. So only due to outer for loop the algorithm becomes linear ie O(n) (suppose n is the size of the array).

Worst case complexity is $O(n^2)$ and it occurs when the input array is reversely sorted. Because in this case always the left side element is greater than the right side element the inner while loop will execute every time. So considering both outer for loop and inner while loop the algorithm becomes quadratic ie $O(n^2)$.

Average case complexity is $O(n^2)$ and it occurs when the input array is randomly sorted. In this case the inner while loop will execute randomly still making the algorithm $O(n^2)$ complexity.

Insertion sort algorithm is simpler and better for very small amount of data. Since the over all time complexity of this algorithm is $O(n^2)$, for large number of input data this algorithm is not good for practical use.

Inversion: Considering the array [16, 3, 5,9,2,11,6,1] we want to find out the number of inversions the unsorted array have.

There are total 18 no. of inversions: 16-3, 16-5, 16-9, 16-2, 16-11, 16-6, 16-1, 3-2, 3-1, 5-2, 5-1, 9-2, 9-6, 9-1, 2-1, 11-16, 11-1, 6-1.

The number of element movements/swaps of the insertion sort is equal to the number of inversions of the array.

MERGE SORT:

The merge-sort algorithm applies the divide and conquer technique to the sorting problem. Divide the input data into two subsets and recursively solve the sub-problems associated with subsets and then take the solutions to the sub-problems and "merge" them into a solution to the original problem.

For all cases complexity of Merge sort algorithm is O(nlog n).
This algorithm needs extra linear memory for the temporary storage of merged data and extra work for copying to the temporary array and back. For this reason this algorithm is not used for sorting data in the computer memory (this algorithm is called external sorting).

We can apply General Divide and Conquer theorem to calculate complexity of merge sort algorithm. We know that $T(n) = aT(n/b) + O(n^k)$ whrer n=size of main problem, a = number of sub-problems, n/b= size of sub-problems and $O(n^k)$ is the complexity of dividing and merging operation.

T(n) is $O(n^{\log_b a})$ if $a>b^k$

T(n) is $O(n^k \log n)$ if $a=b^k$

T(n) is $O(n^k)$ if $a<b^k$

In merge sort the main problem is divided into two sub-problems, so a=2. Size of main problem n is divided by two, so b=2.
Time complexity for dividing is O(1) (constant) and time complexity for merging is O(n) (linear). So total time complexity for dividing and merging operation is O(1) + O(n) = O(n) =$O(n^1)$. So k=1.

Since $a=b^k$ the complexity of merge sort is $O(n^k \log n)$ ie. O(nlog n) (since k=1).

Example: Consider the following Divide and Conquer algorithm:

```
Public long foo(long w, int x, int y, long[] a){
    if(x = = y)
       return a[x];
    int z= (x+y)/2;

    if(a[z] >= w)
        return foo(w,x,z,a);
   else
       return foo(w,z+1,y,a);
   }
```

What is the time complexity of the divide and merge operations? Use Big-Oh notation.

Time complexity for divide operation is constant ie. O(1) because execution of code z=(x+y)/2 takes a constant amount of time. There is no time required for merge operation. So overall time complexity for divide and merge operation is O(1).

How many sub-problems are solved in this algorithm?

Only one sub-problem is solved in this algorithm because in this algorithm the method foo() is called recursively only once either from “if(a[z] >= w)” statement or from “else” statement.

What is the size of the sub-problems? Assume that the problem size n = y-x.

The size of the sub-problem is (y-x)/2 ie. n/2 because the original problem size (y-x) is split into two sizes of (z-x) and (z+1 – y) where z =(x+y)/2.

Apply the General Divide and Conquer Theorem to this algorithm to obtain an estimate of the time complexity.

Here the number of sub-problems, a=1
The size of sub-problem is n/2; b=2
Time complexity for divide and merge operation is $O(1) = O(n^0)$; k=0
Since $a=b^k$, the time complexity of the algorithm is $O(n^k \log n) = O(\log n)$ (since k=1).

QUICK SORT:

Average case and best case complexity is O(n log n)

Worst case complexity is $O(n^2)$

Selection of pivot is very important because depending on the selection of pivot we get worst case or best case time complexity.

If the pivot is selected as the first element or the last element from the array we will get time complexity of $O(n^2)$.

Quick sort is faster than Merge and Heap sort.

Unlike Merge sort, this algorithm does not need extra linear memory and extra work for copying to the temporary array and back.

The algorithm can be used for sorting data in the computer memory (this algorithm is called internal sorting).